Page Allocation Policy

- Last time, covered a number of page replacement policies
  - Handle case when a page must be evicted to free up a frame
- The operating system requires some number of frames…
  - For kernel code and data, I/O buffers, frames for use in interrupt handlers, etc.
- All remaining frames can be allocated to processes, but the OS must do this in an intelligent way
- The operating system can usually control two things:
  - How many frames are allocated to each process
  - The degree of multiprogramming: i.e. how many processes are currently running on the system
- The page allocation policy determines how many page frames each running process should be given
Page Allocation Considerations

• The primary goal of the page allocation policy: make sure all processes have “enough” frames to perform their tasks

• Ideally, every process has all the frames it needs
  • Page faults only occur when a process begins accessing new data that isn’t already in virtual memory

• Reality: try to keep the page fault rate to a reasonable level, so that too much time isn’t lost on paging I/O

• When a given process page-faults, it slows down that process…

• In the context of multitasking, page faults aren’t so bad:
  • When one process faults, it becomes blocked on I/O, allowing other ready processes to have the CPU
  • As long as I/O due to page faults doesn’t become too large, the OS can continue to keep the CPU 100% utilized
Thrashing

- If the amount of paging I/O grows so high that it begins to impede system performance, the system is **thrashing**
  - This term comes from when tapes were used for external storage
  - Describes the sound they would make when paging I/O was high
- An individual process can also be described as thrashing when it has a high page-fault rate
- Thrashing occurs when the OS over-commits the available page frames to running processes
  - The total number of page frames that all processes are using, exceeds the total number of frames available in the system
- Every time a process generates a page fault, the OS evicts a page that some process is still actively using…
  - …which will provoke yet another page-fault again in the near future
Thrashing (2)

• Thrashing can occur unexpectedly as the resource requirements of processes change over time…

• Example: an OS running on a system with 20 physical frames available for applications to use
  • Four processes are running: each has a virtual address space of 10 frames, but each is only accessing 5 of those frames regularly
  • Requirement: 20 frames. The system won’t thrash.

• Then, two processes switch to accessing all 10 frames
  • Now the requirement is for 30 frames, but the system only has 20
  • The I/O overhead from page faults may become so high that CPU utilization may drop precipitously

• This scenario can cause all processes’ page-fault rates to increase, even the ones still only using 5 pages…
Faults and Degree of Multiprogramming

- Every process has a set of pages it is currently using
  - Called the **working set** of the process
- The degree of multiprogramming is the number of processes currently running on the system
- Generally, as we increase degree of multiprogramming, the demand for physical page frames also increases
  - The page fault rate also increases, and the I/O system must handle an increasing rate of page faults
- Eventually the I/O system is saturated, producing thrashing
Degree of Multiprogramming (2)

• Some OSes can control the degree of multiprogramming
  • This is the role of long-term and medium-term scheduling
  • Long-term scheduling controls when new jobs are admitted into the system, and is usually part of batch-processing systems
  • Medium-term scheduling allows the OS to adjust the degree of multiprogramming by suspending running processes, then later resuming them

• Medium-term scheduling can be guided by page-fault rates
  • If system has too high a page-fault rate, remove some processes from memory to free up more frames
  • Later, reintroduce these processes when page-fault rates are lower
Degree of Multiprogramming (3)

- Most widespread operating systems simply rely on the user to control the degree of multiprogramming
  - OS only includes a short-term scheduler
- OS allows the user to start whatever processes they want
- If performance becomes unacceptable (often due to thrashing), the user simply terminates some processes
Global vs. Local Allocation Policies

• When a process needs a new frame, OS has two options:
  • A global replacement policy: the OS can acquire the new frame from any process that is currently running
    • Example: Process A causes a page fault. The OS evicts a page from Process B’s set of pages, and assigns the frame to Process A.
    • The number of frames allocated to a given process can change (both grow and shrink) dynamically over time
  • Strength: gives the OS much more flexibility in assigning frames to processes in memory, based on their needs
    • Frames won’t be held onto by a process that isn’t using them
  • Limitation: a process’ page-fault rate can be directly affected by other processes in the system
    • The performance of a given program may vary widely over time, simply due to the other programs running on the system
Global vs. Local Allocation Policies (2)

- A **local replacement** policy: the OS will acquire the new frame from the process that suffered the page-fault
  - Example: Process A causes a page fault. The OS evicts a page from Process A’s set of pages.
  - Generally, the set of frames assigned to a given process is much more static; it changes in much more limited ways over time
- Strength: if some process starts page-faulting frequently, it won’t affect other processes nearly as much
- Limitation: number of frames assigned to each process isn’t nearly as finely tuned as it is with global allocation

- Most operating systems use a global replacement policy
Simple Page Allocation Policies

- Some page allocation policies are very simple
  - e.g. given a total of \( m \) frames available to \( n \) processes

- An **equal allocation policy** assigns each process roughly \( m / n \) frames

- Some programs have a larger virtual address space than others – equal allocation doesn’t always make sense…
  - e.g. a given process \( p_i \) has a virtual address space of size \( s_i \)
  - The sum of all process virtual memory sizes is \( S = \sum s_i \)

- A **proportional allocation policy** assigns each process a number of frames proportional to its virtual memory size
  - Each process is assigned roughly \( s_i / S \times m \) frames

- These policies make most sense with a local replacement policy; each process has basically static frame allocation
Simple Page Allocation Policies (2)

- As degree of multiprogramming increases, each process ends up with fewer and fewer frames to use
  - e.g. equal allocation policy: each process gets $m / n$ frames; $n$ is the degree of multiprogramming
  - e.g. proportional allocation policy: the sum of all process virtual memory sizes $S$ increases as degree of multiprogramming increases; each process gets roughly $s_i / S \times m$ frames
- Frames for new processes must come from somewhere... unfortunately, that’s all the other processes
- In these systems, medium-term scheduling can be employed to eliminate thrashing
  - Suspend some processes until thrashing issues disappear
  - Resume these processes later when system load is lower
The Working Set Model

- Previous strategies aren’t designed to handle variations in process page-frame requirements
- Would prefer to determine the requirements of each process much more dynamically
- One option: quantify the size of each process’ working set
  - The set of pages that the process is currently working with
  - As a program runs, its working set changes
- A process’ working set is difficult to determine...
- Estimate it by looking at all recent memory accesses within a given window
  - A proper choice of \( \tau \) is essential for a good estimate
The Working Set Model (2)

- Can estimate a process’ working set with a mechanism very similar to the Aging page-replacement policy
  - Requires that the MMU maintains an “accessed” bit for each page
- The OS maintains a $b$-bit value for each page
- On a periodic timer interrupt, the OS traverses all pages in memory, updating this value
  - Shift the value right, store “accessed” bit into topmost bit of value, then clear the page’s “accessed” bit
- If a process’ page has a nonzero value, it’s in the process’ working set
  - Can easily count how many pages are in each process’ working set
- Main difference between this and the Aging policy is that the timer interval is much longer
  - e.g. might only want 2-4 interrupts during the working set window
The Working Set Model (3)

- Once each process’ working set is known (or guessed), OS can use this to set the degree of multiprogramming
  - If the working sets of all processes requires fewer frames than the system has available, add more processes!
  - Or, if the working sets of all processes don’t fit within the system’s memory, suspend processes until all working sets fit within memory
  - (Can suspend lower priority processes to allow higher-priority ones to complete more quickly)

- This approach works very well in practice

- This model also does a great job of driving prepaging of processes when they are unblocked or resumed
  - Instead of waiting for pages to be demanded, preemptively start loading a process’ working set back into physical memory
  - If working-set estimate is good, should greatly reduce page faults
Page-Fault Frequency

- So far, these page allocation policies primarily make sense in the context of medium-term scheduling
  - Operating systems that can control degree of multiprogramming
- Most widespread operating systems don’t actually have a long-term or medium-term scheduler
- Instead, they rely on the **page-fault frequency** of various processes to determine when to assign more frames
  - i.e. how often is a given process generating page faults?
- If a process’ page-fault frequency is too high, give it more frames
  - Take frames away from processes with a low page-fault frequency
  - Premise: they may have more frames than their working set size
Page-Fault Frequency (2)

- The pager can aim to keep processes’ page-fault frequency within a specific “desirable range”
  - Specify a lower- and upper-bound on page-fault rate
- If a process’ page-fault rate exceeds the upper bound, the OS assigns it more frames
- If a process is below the lower bound, the OS considers it a candidate for giving up frames
  - e.g. with page buffering, OS might reclaim frames and move them into the free page-frame pool
- If the process is in the desirable page-fault frequency range, the OS can use other page replacement policies
  - e.g. LRU, aging, etc.
  - This can be done in a local way, to avoid affecting other processes
Page-Fault Frequency (3)

- Of course, using page-fault frequency approach won’t necessarily prevent thrashing...
- The OS may still benefit from medium-term scheduling techniques to free up more frames for processes
  - e.g. if page-fault frequency of many processes is high, just suspend one or more processes and use their frames for other processes
Modern OSes and Page Allocation

• Most widely used operating systems don’t have a very sophisticated page allocation policy
• Instead, rely on a more sophisticated page replacement policy with global replacement
• Let users run the programs they want to run
  • When system performance become unacceptable, the user will kill or exit some of the running processes
  • (The user is the medium/long-term scheduler…)
• If a process remains blocked for a long time (e.g. waiting for user interaction):
  • If other processes need more frames, will be taken from this one
  • The process will eventually be completely paged out of memory
  • All of the process’ frames will become available to other processes
Example: Linux

- Linux has a straightforward virtual memory policy:
  - Pages are never allocated to a process until they are used
    - e.g. programs are referenced in the process’ virtual memory mapping, but are demand-paged into physical memory
    - e.g. heap and stack pages aren’t allocated to a process until it actually references the page
      - **Demand-zero paging**: a page is taken from the free-page pool and cleared with all zeros; saved in swap space for anonymous memory
  - Linux uses a modified form of the Clock algorithm that maintains an age for every page
    - Pages are periodically traversed, and ages adjusted based on whether or not they have been accessed recently
      - Ages decay to zero; higher ages indicate frequent recent accesses
    - Pages with a low age value will be reclaimed by the system
    - Allows Linux to implement a policy that considers both recency and frequency of page accesses
Example: Windows

- Windows generally relies on processes to state their “working set” sizes
  - Windows calls it the “working set,” but it’s actually the **resident set**; the number of pages the process has in physical memory
  - If the process’ actual working set (not the Windows definition) is larger than its resident set, the process will incur many page-faults
- Processes are given a default minimum and maximum “working set” size
  - e.g. default minimum is 50 pages or 200KiB, default maximum is 345 pages or ~1.3MiB
  - A process can have a larger virtual address space than this…
- Processes may also specify their minimum and maximum “working set” sizes
  - Limits are applied, e.g. a hard minimum of 20 pages, etc.
Example: Windows (2)

- If the page-fault rate becomes unacceptably high, Windows will **trim** the “working sets” of all processes
  - (Again, Windows “working set” means the resident set)
- When trimming, Windows sets goal for page reclamation
  - Sets it slightly higher than what is strictly required, so that trimming doesn’t have to run multiple times
- Pages in each process’s working set are periodically aged based on recent accesses
  - In Windows 7, each page has one of up to 8 age values
- Oldest pages are reclaimed first to avoid further faults
- Trimming culls pages from all processes, even if trim goal is reached partway through the procedure
  - Ensures that page reclamation is fair to all processes
Example: Windows and Linux

• Both Windows and Linux use free page-frame pools
• When pages are reclaimed by the virtual memory system, they are added to clean or modified pools
  • Modified pages are eventually written out, then moved to the clean pool
• If a process page-faults on a page in one of these pools, the page will usually be reassigned back to the process
Next Time

• Start discussing filesystems